AMENDMENTS TO THE SPECIFICATION:

Please amend the first paragraph on page 1 as follows:

A1

The present application is related to U.S. Patent Application No. 08/956,717, filed on October 23, 1997, to Choi et al., entitled "DETERMINISTIC REPLAY OF MULTITHREADED APPLICATIONS", now U. S. Pat. No. 6,101,524 ~~having IBM Docket No. YO997-355~~, and to U.S. Patent Application No.09/569,308, filed on May 11, 2000, to Choi et al., entitled "METHOD AND APPARATUS FOR DETERMINISTIC REPLAY OF JAVA MULTITHREADED PROGRAMS ON MULTIPROCESSORS" ~~having IBM Docket No. YO999-501~~, both assigned to the present assignee, and incorporated herein by reference.

Please amend the third paragraph on page 10 as follows:

A2

Figure 3B illustrates a general critical (GC)-critical section 300 for general critical events;

Please amend the third paragraph on page ~~23~~ as follows:

A3

The logical thread schedule of the present invention as described above preferably is generated using a global counter and a local counter. An example of the operations in capturing the logical thread schedule is illustrated in FIG. 3A. Exemplary logical thread schedules and how they are identified is illustrated in FIG. 4.

Please amend the second paragraph on page 24 as follows:

A4

There are many programs and methods for generating a logical thread schedule including that shown in Figure 3A. It is note that the "global clock" and "global counter" are synonymous. That is, an example of the operations in capturing the logical thread schedule is illustrated in FIG. 3A ~~3~~. The operations begin in step 301 by initializing the variable GLOBAL_CLOCK to 0. GLOBAL_CLOCK is shared by all threads generated by the physical thread scheduler during the execution of the program. Such threads are denoted as thread[1] through thread[last]. The

physical thread scheduler creates such threads in a serial fashion. The software tool of the present invention deterministically assigns a thread identifier to the newly created thread and passes the thread identifier to the newly created thread. The thread identifier is initialized to 0 and incremented automatically for each thread created. For each thread, a thread-specific logical thread schedule is computed (steps 303 through 315). The figure shows the steps for thread[i] in detail as a representative case.

Please amend the first paragraph on page 27 as follows:

Thus, each critical event is uniquely associated with a global counter value. Global counter values in turn determine the order of critical events. Therefore, updating the global counter for a critical event and executing the critical event, are performed in one atomic operation for shared-variable accesses. Some synchronization events are handled differently to avoid deadlocks (e.g., for a detailed description, see the above-mentioned Jong-Deok Choi and Harini Srinivasan, "Deterministic replay of java multithreaded applications", Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pages 48-59, August, 1998), the present inventors have implemented light-weight GC-critical section (e.g., Global Counter critical section) codes to implement a single atomic action of critical events by guarding them with *GcEnterCriticalSection* and *GcLeaveCriticalSection*, as shown in steps 301-305 of Figure 3B. It is used when the critical event is a general event (e.g., a shared variable access). *GcEnterCriticalSection* and *GcLeaveCriticalSection* are implemented by acquiring and releasing a light-weight lock (e.g., an "efficient," lock, which does not incur much runtime overhead) called *GCounter__Lock*. Synchronization events with blocking semantics, such as **monitorenter** and **wait**, can cause deadlocks if they cannot proceed in a GC-critical section. Therefore, the invention handles these events differently by executing them outside a GC-critical section (e.g., for a detailed description, see the above-mentioned Jong-Deok Choi and Harini Srinivasan, "Deterministic replay of java multithreaded applications", Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pages 48-59, August, 1998).

Please amend the first paragraph on page 35 as follows:

In step 1005 on the client side, a step of record critical event occurs. This step includes steps 1005A ~~1001A~~ of enterGCCriticalSection, update ClientGC, and leaveGCCriticalSection. It is noted that step 1007 and step 1007B on the server side is substantially similar to that of steps 1005 and 1005A ~~1001~~ on the client side (e.g., step 1007 is performed when the server encounters a critical event on its side.

Please amend the first paragraph on page 37 as follows:

Figure 8(a) shows the process during the record mode for *read()*. In step 801 ~~1801~~, the *read* event is executed, returning "n", the number of bytes read which is logged in recordedValue in step 802 ~~1802~~. The critical event corresponding to the read is logged in step 803 ~~1803~~ and the process exits in step 804 ~~1804~~. Step 803 ~~1803~~ is essentially 803A ~~1803A~~ which involves entering the GC critical section, updating the global counter and leaving the GC critical section.

Please amend the fifth paragraph on page 38 as follows:

A solution is to just record the occurrence of such an event and allow other unrelated events (i.e., events that do not operate on the same socket) to proceed. Events that do use the same socket will be blocked by using a lock variable for each socket. This is shown in Figures 12(a) and 12(b) respectively. That is, steps 1201-1204 in Figure ~~Figures~~ 12(a) and steps 1251-1254 of Figure 12(b) respectively illustrate ~~an~~ exemplary steps ~~flowchart~~ for implementing a more efficient record and replay code of a read and a record and replay code for a write. The enterFDCriticalSection(socket) in step 1201 (1251) of figures 12(a) and 12(b) ensures that only *reads* or *writes* corresponding to that socket execute the code inside.